<u>Amendments to the Specification</u>

Amend the paragraphs in the specification as follows:

[0002]The present invention relates to data processing systems and, more particularly, to a process for producing compressed ~~Java~~ <u>JAVA™</u> Jar files including byte code which remains directly interpretable by a ~~Java~~ <u>JAVA</u> virtual machine.

[0004]Programs written in the ~~Java~~ <u>JAVA</u> programming language are compiled into class files and are typically grouped together by placing them in an archive file known as a ~~Java~~ <u>JAVA</u> Archive (Jar) file. These Jar files contain all of the class files generated by compiling the application source code. When the applications are interpreted many additional classes are typically required. These classes come from the ~~Java~~ <u>JAVA</u> runtime library. For embedded systems, such as television set top boxes, cell phones, PDA's and the like that support the interpretation of ~~Java~~ <u>JAVA</u> code, the classes that make up the ~~Java~~ <u>JAVA</u> runtime library are often built into the device and reside in some form of non-volatile memory such as Flash memory. Applications, on the other hand, are downloaded into the device over a network at the time the user wishes to execute them.

[0005]A problem faced by embedded systems developers is that many of these devices have severely constrained memory resources, and it is a serious challenge to get the required ~~Java~~ JAVA libraries and applications to fit into available memory. This problem is exacerbated by the fact that ~~Java~~ <u>JAVA</u> class files are large and not particularly efficient representations. ~~Java~~ <u>JAVA</u> employs runtime binding which requires that the class files contain a great deal of symbolic information in the form of strings. When a group of class files are combined into a Jar file, there will be many redundant copies of the same string information.

[0006]Two techniques are in common usage to combat this problem: jar file compression, and obfuscation. The jar file format supports the use of zip type data

compression on individual files within the jar. Such compressed jars are typically on the order of 50% of the size of an uncompressed jar. The drawback of using compressed jars is that they cannot be executed without first decompressing them. This means that the device must have enough memory available to hold the compressed jar, in addition to uncompressed copies of each referenced class. Hence such compressed jars are useful for reducing the bandwidth required to transmit an application jar over a network, but do not really help with reducing device memory requirements. Obfuscation involves the automated modification of class, method, and field names from the original names employed by the application developers to arbitrary names. Obfuscation was originally employed to make reverse engineering ~~Java~~ JAVA applications more difficult since the arbitrary names make it much harder to understand the code. However, if the arbitrary names are made as short as possible, the application Jar is reduced in size. The reduction in size achieved by traditional obfuscation can be on the order of 15% to 20% for large applications. Unlike compressed jars, obfuscated jars are directly executable. The small size reduction achieved, however, is of limited use.

[0007]A method for processing a ~~Java~~ JAVA Archive (JAR) file to provide an application file adapted for a target environment in accordance with one embodiment of the invention comprises removing from the JAR file at least a portion of information not necessary for running the application; mapping at least one of application defined class, field and method names to shorter names; and mapping at least one of target environment defined class, field and method names to corresponding target device names.

[0013]The subject invention will be described within the context of a process denoted as "Grinding," in which a number of techniques are employed to produce compressed ~~Java~~ JAVA jar files that remain directly interpretable while achieving a significant reduction in size. In addition, all class, method and field bindings remain symbolic, thereby preserving polymorphism. The Grinding process transforms ~~Java~~ JAVA jars into a format known as a "ground" jar. Since a ground jar is in a different format than a

~~Java~~ JAVA jar, it cannot be interpreted by a standard ~~Java~~ JAVA Virtual Machine (VM), but only by a virtual machine that has been "grind enabled." If desired, however, the grind process can be reversed by an "ungrinding" process, which converts the ground Jar file into a conventional ~~Java~~ JAVA jar file containing conventional ~~Java~~ JAVA class files which may be interpreted by standard ~~Java~~ JAVA VM. The subject invention will be illustrated within the context of a client server environment in which a server is operative to provide video or other information to a client (e.g., a set top box) via a network (e.g., a cable, telecommunications, satellite or other network). Within the context of the present invention, the term executable is defined as the interpretation of byte code by a ~~Java~~ JAVA virtual machine (VM) and not execution of machine code by a central processing unit (CPU).

[0017]The inventors contemplate that the invention may be segmented into a server function and a client function. The server function comprises, e.g., the grind method and tools for performing the grind process, which may be implemented on the server 108 to produce a ground jar file for propagation via the network 106 to the client presentation engine 104. The client function comprises, e.g., the ~~Java~~ JAVA virtual machine (VM) environment and general application target environment (i.e., platform) that interprets the ground Jar file to execute the application. These functions will be discussed in more detail below with respect to the figures and the tables included herein. The functions may be implemented as a method by one or more processors. The functions may be embodied as software instructions within a signal bearing medium or a computer product. Within the context of a peer to peer network, the server functions and client functions may be both be implemented on client and/or server devices.

[0026]FIG. 3 depicts a flow diagram of a method according to an embodiment of the present invention. Specifically, method 300 of FIG. 3 comprises the processing of a ~~Java~~ JAVA jar file to provide a ground jar file. In the embodiment of FIG. 3, the steps employed are listed in a particular sequence. However, the steps of the grind process may be

invoked in various other sequences and such other sequences are contemplated by the inventors.

[0027]The Grind process of the present invention may be performed by, for example, a server or other computing device implemented using the controller topology 200 discussed above with respect to FIG. 2 or other computing topology. The Grind process 300 of FIG. 3 employs the following techniques during the transformation of a ~~Java~~ JAVA jar to a ground jar: (step 305) receiving a ~~Java~~ JAVA Jar file for processing; (step 310) invoking an archive tersing method; (step 320) invoking a class tersing method; (step 330) invoking an opcode replacement method; (step 340) invoking an unreferenced member culling method; (step 350) invoking a string pooling method; (step 360) invoking a constant pool optimization method; and/or (step 370) invoking an obfuscation method and (step 380) providing a resulting ground Jar file. Each of these techniques will now be discussed in more detail.

[0029]A ~~Java~~ JAVA jar file typically includes information that is not required in or critical to an embedded environment, including multiple copies of file names, file modification date and time stamps, CRC's for each file, file access permissions and the like. Archive tersing transforms the archive into a much simpler format that includes only information required for a particular application. An exemplary format of a ground archive is shown below:

[0033]Class tersing involves transforming each class file in the ~~Java~~ JAVA jar into a format having a reduced size. Within the context of class tersing, one or more of the following class file structures modifications are provided: (a) removal of class attributes; (b) changes to class field attribute structure; and (c) change to method attribute structure.

[0037]If all of the above changes to the class file structures are implemented, there will be several new limits to the underlying ~~Java~~ JAVA code. Specifically, the maximum local variables per method is reduced to 255 from 65535, the maximum stack per method is reduced to 255 from 65535 and the maximum handlers per method is reduced to 255 from 65535. The maximum code size per method is unchanged.

[0038]It is noted by the inventors that the above limitation will not impact most ~~Java~~ JAVA code. For example, the Multimedia Home Platform (MHP) extension of the European Digital Video Broadcasting (DVB) standard utilizes ~~Java~~ JAVA runtime libraries consisting of approximately 2278 classes with 8096 fields and 18347 methods. Over this large body of code, the closest any limits were approached were: max local variables per method 40, max stack per method 18, max handlers 30, max code size 7552.

[0040]The ~~Java~~ JAVA compiler converts the ~~Java~~ JAVA language source code into byte codes (binary representation of the low level instruction set of the ~~Java~~ JAVA Virtual Machine). Opcode replacement involves replacing certain byte codes generated by the compiler with more compact versions. The ~~Java~~ JAVA Virtual Machine instruction set was examined by the inventors for opportunities for size reduction, and a large body of compiled ~~Java~~ JAVA code was analyzed for instruction usage statistics.

[0041]A set of "short" byte codes was created by the inventors to exploit the opportunities for size reduction. These byte codes use the range reserved by the ~~Java~~ JAVA Virtual Machine Specification for the set of "quick" byte codes. This range was selected since these byte codes are never generated by a ~~Java~~ JAVA compiler, they are reserved for the internal use of ~~Java~~ JAVA virtual machines. These short byte codes achieve size reduction by reducing constant pool indices from 16 bits to 8 bits, reducing relative branch offsets from 16 bits to 8 bits, and reducing switch offsets from 32 bits to 16 bits. Exemplary Opcode Replacements are disclosed below with respect to Table 1. It will

be appreciated by those skilled in the art that various modifications to the actual replacement codes may easily be made while still practicing the invention.

[0043]The grind process tracks references to methods and fields across the entire input jar and those methods and fields that are found to have zero references become candidates for removal. However, even though a field or method is not directly referenced within an archive, it might still be required. One example of this is with the ~~Java~~ JAVA runtime libraries built into an embedded device. A great many methods are unreferenced internally within the ~~Java~~ JAVA runtime, but they need to be available for applications to use. Unreferenced methods that are private may be removed from the ~~Java~~ JAVA runtime.

[0044]For an application, too, it is possible for a method to appear to be unreferenced, but actually still be required. Examples of this are application methods invoked by the ~~Java~~ JAVA runtime library such as the applet lifecycle methods. Also, methods that implement interfaces may be invoked by the ~~Java~~ JAVA runtime. Finally, a method that overrides a superclass method may be referenced by code that treats the object as an instance of the superclass, and hence the reference will appear to be of the superclass method implementation, not the subclass implementation.

[0045]In one embodiment, unreferenced static final field declarations are removed. Static final field declarations primarily exist for the benefit of the ~~Java~~ JAVA compiler. All references to static final fields of non-object type, or String object type are in-lined at compile time. The field declarations of these fields are not required at runtime.

[0046]In another embodiment, ConstantValue attributes from final fields are removed. The ~~Java~~ JAVA compiler will always generate code to initialize these fields by directly referencing the constant initializer in the constant pool. The ConstantValue attribute in the field declaration is not required at runtime.

[0049]String pooling involves creating a single table of strings for the entire jar. This eliminates the duplication of strings from class to class. String constants are illustratively stored in ~~Java~~ JAVA class files as Utf8 constant pool entries. The grind process determines the set of unique Utf8 strings across the jar and stores them in a single table. It then removes all Utf8 string entries from each class's constant pool, and replaces all references to the removed strings with references to the common string table. References to the strings are in the form of indexes into class constant pools, so these indexes are mapped to indexes into the common string table. Since the indices used to reference strings are 16 bits in size, this limits the maximum number of unique strings in a ground jar to 65536. As with class tersing, this limit is not often exceeded. Using MHP as an example again, the 2278 classes contain only 13649 unique strings.

[0052]When constant pool optimization is used within the grind process, the constant pool of each class in the ~~Java~~ JAVA jar has one or more of the following operations performed on it: (a) Remove unreferenced entries; (b) Make entries a fixed size; and (c) Sort entries by type.

[0054]A problem with the ~~Java~~ JAVA class file format is that information is referenced from the class constant pool by index, but the constant pool entries are of variable size. This means that the ~~Java~~ JAVA Virtual Machine must allocate memory to create a table that maps constant pool indices to constant pool entries. The grind process converts constant pool entries to a fixed size thereby allowing the virtual machine to directly access constant pool entries by index. A fixed size of 4 bytes per entry is used. This is made possible by moving all Utf8 strings from the constant pool to a common string pool, and by removing the 8 bit type field from pool entries. Thus all pool entries fit in 4 bytes except for Long (64 bit integer) and Double (64 bit float) entries, which require 8 bytes, and are allowed to occupy two consecutive pool entries.

[0073] The grind process performs two types of obfuscation, Application Obfuscation and Platform Obfuscation. Application obfuscation maps application defined class, field, and method names to shorter names. Platform or target environment obfuscation maps class, field, and method names for classes built into the target device (for example all the ~~Java~~ JAVA runtime library classes, and vendor specific ~~Java~~ JAVA libraries built into a cell phone, personal digital assistant (PDA), set top box and the like). Specifically, references in applications to class, field, method names and, more generally, symbols in the target environment get mapped to the same shorter name that was used when processing the target environment. For example, if the symbol "java.lang.Runtime" becomes "#" after grinding, then in every place where "java.lang.Runtime" is found in any application the grind process will replace the reference with "#". This is the consistency rule that is maintained in order for ground applications to run correctly.

[0074] Since the ~~Java~~ JAVA libraries built into the device are known by the device vender but many applications may be built by many independent developers, a name mapping scheme that maintains a consistent mapping of platform or target environment names and independent mappings for application names is required.

[0092]To grind an application for a particular ~~Java~~ JAVA environment it is necessary to provide an obfuscation library for the target environment as well as the ~~Java~~ JAVA jar of the application. The invention is applicable to various bytecode interpretation environments such as the Liberate TV-Navigator middleware provided by Liberate Technologies, Inc. of San Carlos, Calif.

[0093]The obfuscation library for the target environment is created using the statdb tool. Statdb analyzes all of the ~~Java~~ JAVA code supplied by the target environment and sample applications that will be run on the target, and generates an optimal name mapping scheme for obfuscation. This scheme is stored in the target platform or target environment obfuscation library. For Liberate's TV Navigator, this library is called Classes.lib. Once generated, the target platform or target environment obfuscation library is incrementally updated using libdb as the target platform or target environment is modified. This allows a backwards compatible name mapping to be maintained as the target platform or target environment ~~Java~~ JAVA runtime evolves over time.

[0096]The target environment or platform may comprise, for example, a resource constrained device such as a cell phone, personal digital assistant (PDA), set top box and the like. The ~~Java~~ JAVA libraries built into the device are known by the device vendor. A name mapping scheme maintains a consistent mapping of platform names and independent mappings for application names such that the grinding of a Jar file provides the appropriate naming of class, field, method names and the like. In this manner, efficient target device interpretation of the ground Jar application is enabled. In various embodiments of the invention, "standard" classes, fields, methods and the like are preferentially replaced by target device specific classes, fields, methods and the like such that optimizations adapted for the target environment are made during the grinding process. The target device receives a ground Jar file and iteratively resolves application defined and target environment defined class, field and method names to interpret thereby

application byte codes presented within a ground Jar file. In this manner, the application is executed by the target device.

[0097]The target environment includes an interpretation engine that recognizes and interprets ground ~~Java~~ JAVA bytecode read from ground classes in the ground jar file. The target environment is implemented in a manner allowing it to parse the ground class file and jar formats (such as noted above with respect to step 370). The target environment is capable if interpreting the ground jar file because only the unnecessary portions were removed leaving the essential portions intact. The target environment uses the obfuscated symbols directly (rather than attempting to convert them back to their original names). This is possible because the grind process ensures that applications use the same symbol name mapping (from original name to obfuscated name) for all target environment symbols." For example, as discussed above with respect to step ??, the symbol "java.lang.Runtime" becomes `v` after grinding such that every reference to "java.lang.Runtime" in the application is replaced with a reference to "#".

[0098]The ungrind process is the process that is used to reverse the grind process. That is, the entire grind process is reversible such that ground jar files can be reverted back into conventional ~~Java~~ JAVA class files with unobfuscated names. In one embodiment, the ungrind process comprises the following steps: (1) Using the obfuscation library to map from obfuscated name back to the original names (both for application and target environment symbols); (2) Padding out the truncated class file fields to their original size; (3) Inserting default values for any fields that were removed; (4) Reinserting strings from the string pool back into the appropriate class files; and (5) Restoring the class constants by reinserting their type information. The ungrinding process makes it possible for ground jars to be interpreted correctly on standard (non-grind enabled) ~~Java~~ JAVA VM environments.